Тестирование React приложений

В этом уроке, мы рассмотрим следующие темы:

1. Введение: Зачем тестировать и Пирамида тестирования.
2. Экосистема: Сравнение Jest и Vitest, отказ от Enzyme.
3. Настройка окружения и запуск тестов.
4. Основы синтаксиса Jest: Describe, Test, Expect.
5. Философия React Testing Library и работа с DOM.
6. Тестирование приложения через RTL.
7. Тестирование асинхронности и взаимодействие с API.
8. Техники мокирования (Mocking) данных и функций.
9. Лучшие практики (Best Practices) и частые ошибки.

Введение: Зачем тестировать и Пирамида тестирования

Прежде чем мы начнем писать код и устанавливать библиотеки, важно понять «философию» тестирования. Многие разработчики пропускают этот этап, считая, что написание тестов - это просто трата времени, которое можно было бы потратить на новые фичи. Сегодня мы разберем, почему это не так.

Зачем вообще тестировать React-приложение?

Представьте, что мы разрабатываем приложение Wildland - большую платформу для путешественников. В ней сотни компонентов: карточки туров, формы бронирования, фильтры поиска.

Если вы не пишете тесты, вы все равно тестируете приложение - просто вы делаете это вручную. Вы кликаете по кнопкам, заполняете формы и смотрите, не сломалось ли что-то. Проблема в том, что ручное тестирование медленное, ненадежное и его невозможно масштабировать.

Автоматические тесты решают следующие задачи:

1) **Уверенность при рефакторинге** (главная причина). Вы захотели оптимизировать код компонента поиска в приложении. Без тестов вам страшно менять старый код («Работает - не трогай!»). С тестами вы меняете код, запускаете команду проверки и через 5 секунд знаете: сломали вы логику поиска или нет.

2) **Отлов багов на ранней стадии**. Исправить баг в момент написания кода стоит 5 минут. Исправить баг, когда он уже ушел в продакшн и о нем сообщили пользователи - стоит часы или дни (диагностика, фикс, деплой).

3) **Живая документация**. Новый разработчик приходит в проект. Как понять, что делает функция calculateTripPrice? Он смотрит тест и видит: «Ага, если передать 2 взрослых и 1 ребенка, цена должна быть X». Тесты никогда не врут, в отличие от устаревших комментариев в коде.

Пирамида тестирования (Testing Pyramid)

В мире разработки существует концепция «Пирамиды тестирования». Она показывает, каких тестов должно быть много, а каких - мало.

Представьте пирамиду, разделенную на три уровня (снизу вверх):

Уровень 1 (Основание): Unit-тесты (Модульные)

- Что это: Тестирование самых маленьких, изолированных частей кода (функций, утилит, хуков) в отрыве от остального приложения.
- Скорость: Очень быстрые (миллисекунды).
- Стоимость поддержки: Дешево.
- Пример: Проверка функции, которая форматирует дату поездки в Wildland (2023-10-05 -> 5 Oct).
- Количество: Их должно быть больше всего.

Уровень 2 (Середина): Интеграционные тесты (Integration)

- Что это: Проверка того, как несколько модулей работают вместе. В контексте React это чаще всего тестирование компонентов. Мы проверяем не внутренний код функции, а то, что компонент правильно отрисовался и реагирует на действия пользователя.
- Важный нюанс: В современном React (с использованием React Testing Library) граница между Unit и Integration размыта. Мы стараемся тестировать компоненты так, как их видит пользователь.
- Пример: Рендерим форму входа Wildland, вводим логин/пароль, нажимаем кнопку и проверяем, что вызвалась функция отправки данных.
- Количество: «Золотая середина». Именно на них мы сделаем упор в этом уроке.

Уровень 3 (Верхушка): E2E-тесты (End-to-End)

- Что это: Тестирование всего приложения целиком в реальном браузере. Робот открывает Chrome, заходит на сайт и кликает как реальный пользователь.
- Инструменты: Cypress, Playwright.
- Скорость: Медленные (секунды или минуты).
- Стоимость: Дорого писать и поддерживать (они часто ломаются из-за мелочей).
- Пример: Робот проходит полный путь от главной страницы Wildland до успешной оплаты тура.
- Количество: Их должно быть немного. Они проверяют только критические бизнес-сценарии (Happy Path).

«Нулевой» уровень: Статический анализ

Под пирамидой находится фундамент - **TypeScript** и **ESLint**. Они отлавливают опечатки и глупые ошибки (например, попытку умножить строку на число) еще до того, как вы запустите тесты.

Мы не тестируем, чтобы «найти баги». Мы тестируем, чтобы уверенно вносить изменения в будущем. Мы будем ориентироваться на **интеграционные тесты компонентов**, так как они дают наибольшую пользу (value) при разработке интерфейсов.

Пирамида учит нас: пиши много простых быстрых тестов и немного сложных медленных.

Экосистема: Сравнение Jest и Vitest, отказ от Enzyme.

Чтобы написать тест, нам нужны две вещи:

- **Test Runner** (запускатор): программа, которая находит файлы с тестами, выполняет их и говорит: «Тест прошел» или «Тест упал».
- **Testing Utility** (помощник): библиотека, которая умеет "рисовать" (рендерить) React-компоненты в виртуальной среде, чтобы мы могли с ними взаимодействовать.

Битва раннеров: Jest против Vitest

Долгое время королем был Jest. Но с появлением сборщика Vite ситуация изменилась.

Jest – это стандарт индустрии от Facebook. Огромное сообщество, куча плагинов, встроен по умолчанию в Create React App (CRA).

Минусы: может быть медленным на больших проектах. Сложнее настраивать для современных проектов на Vite (требует лишних конфигураций babel).

Когда использовать: В старых проектах, в Next.js (хотя и там есть подвижки), или если используете Create React App.

Vitest - новый игрок, созданный специально для работы в связке с Vite. Невероятно быстрый (использует те же настройки, что и ваш сборщик Vite). Поддерживает ESM (современные модули) из коробки. Он специально сделан совместимым с Jest. Если вы знаете команды Jest (describe, test, expect), вы уже знаете Vitest.

Смерть Enzyme и победа React Testing Library (RTL)

Раньше (до 2019 года) стандартом был Enzyme от Airbnb. Сейчас он считается устаревшим (deprecated) и не поддерживает новые версии React (18+).

Почему от него отказались? Из-за проблемы "Тестирования деталей реализации".

Представьте, у нас есть компонент кнопки, который внутри себя хранит состояние isClicked. Мы писали тесты, которые лезли во "внутренности" компонента: "Проверь, что после клика переменная state.isClicked стала равна true".

Это плохо, потому что:

1) Пользователю плевать на переменную isClicked. Ему важно, изменился ли цвет кнопки или появился ли текст "Спасибо".

2) Если вы перепишете код и переименуете isClicked в hasBeenPressed, тест упадет, хотя для пользователя кнопка работает так же. Это называется хрупкий тест.

Подход React Testing Library (Современный)

RTL проповедует философию - чем больше ваши тесты похожи на то, как приложение использует пользователь, тем больше уверенности они дают. Мы пишем тесты, опираясь на то, что видит пользователь: "Нажми на кнопку. Проверь, что на экране появился текст 'Забронировано'".

Нам всё равно, как это реализовано внутри (useState, Redux или магия). Главное - результат в DOM. Для разработки нашего прилоежния мы выберем современный стек (последняя колонка):

Инструмент	Категория	Статус	Комментарий
Jest	Test Runner	Стандарт	Классика, но медленнее Vitest.
Vitest	Test Runner	**Модерн**	Идеален для Vite-проектов. Быстрый.
Enzyme	Utility	Устарел	Не использовать. Тестирует "кишки" компонента.
React Testing Library	Utility	**Стандарт**	Тестирует поведение пользователя. Идет в комплекте с React.

Мы больше не используем **Enzyme**, потому что не хотим тестировать внутренние переменные (state). Мы хотим тестировать то, что видит пользователь. Для запуска тестов мы используем **Jest** или **Vitest**. Их синтаксис почти одинаков, поэтому, научившись писать тесты на одном, вы сможете работать и с другим.

Настройка окружения и запуск тестов.

Выполним создание проекта:
https://docs.google.com/document/d/1mHXNR8t6La1YKs_eG7E7O-bK5lkSQleV/edit

Прежде чем писать код, убедимся, что наш раннер (Jest или Vitest) установлен и готов к работе с проектом.

1. **Установка Vitest** (или **Jest**). В большинстве современных проектов (Next.js, Create React App) Jest уже установлен. Если нет, установите его как зависимость для разработки:

Команда установки **Vitest** (наш вариант): **npm install -D vitest**

Команда установки **Jest**: **npm install --save-dev jest**

2. **Создание скрипта для запуска**. Чтобы запускать тесты одной командой, необходимо добавить скрипт в ваш файл **package.json**:

Настройка для **Vitest** (наш вариант):

```
// package.json
"scripts": {
    "test": "vitest"
  }
```

Когда в **package.json** прописывается **"test": "vitest",** npm понимает, что при выполнении команды:

```
npm test
```

нужно запустить программу **Vitest**. Без этой строки npm не знает, какой инструмент должен выполнять тестирование, и команда npm test либо выдаст ошибку, либо запустит пустой скрипт по умолчанию.

Настройка для **Jest**:

```
// package.json
{
  "scripts": {
    "test": "jest",
    "test:watch": "jest --watch" // Для автоматического запуска тестов при
изменении файлов
  }
}
```

Файл теперь выглядит следующим образом:

```
{
  "name": "testapp",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint .",
    "preview": "vite preview",
     "test": "vitest"
  },
  "dependencies": {
    "@tanstack/react-query": "^5.90.11",
    "lucide-react": "^0.556.0",
    "prop-types": "^15.8.1",
    "react": "^19.2.0",
    "react-dom": "^19.2.0",
    "react-router-dom": "^7.10.1",
    "swr": "^2.3.7"
  },
  "devDependencies": {
    "@eslint/js": "^9.39.1",
    "@types/react": "^19.2.5",
    "@types/react-dom": "^19.2.3",
    "@vitejs/plugin-react": "^5.1.1",
    "eslint": "^9.39.1",
    "eslint-plugin-react-hooks": "^7.0.1",
    "eslint-plugin-react-refresh": "^0.4.24",
    "globals": "^16.5.0",
    "vite": "^7.2.4"
  }
}
```

3. Создание функционального кода. Напишем простую функцию, которую мы будем тестировать. В папку **src**, добавим файл **priceUtils.js**:

```
const SERVICE_FEE = 50;

/**
 * Рассчитывает общую стоимость, включая сервисный сбор.
 * @param {number} basePrice Базовая цена тура.
 */
export const calculateFinalPrice = (basePrice) => {
  if (basePrice <= 0) {
    return SERVICE_FEE; // Минимальная цена - только сбор
  }
  return basePrice + SERVICE_FEE;
};
```

Вначале мы определили константу **SERVICE_FEE**, которая хранит фиксированную величину сервисного сбора. Затем определяется функция **calculateFinalPrice**, которая принимает базовую цену тура и должна вернуть итоговую стоимость. Внутри функции выполняется проверка: если базовая цена меньше или равна нулю, то функция возвращает только сервисный сбор, потому что минимальная возможная цена состоит именно из него. Если базовая цена больше нуля, то функция складывает её с сервисным сбором и возвращает сумму.

Теперь, давайте проверим, правильно ли эта функция рассчитывает итоговую стоимость тура. Мы будем проверять, добавляет ли она фиксированный сервисный сбор к положительной базовой цене и возвращает ли она только сервисный сбор, если базовая цена равна нулю или меньше.

4. **Написание теста и Соглашения об именовании файлов**. Jest (и Vitest) автоматически ищет файлы, которые соответствуют определенным правилам именования:

1) Ваши тестовые файлы должны находиться в папке __tests__ или рядом с компонентом.

2) Или, что более популярно, заканчиваться на .test.js или .spec.js.

В папке **src/tests**, создадим файл **priceUtils.test.js**:

```
import { describe, it, expect } from "vitest";
import { calculateFinalPrice } from "../priceUtils";

describe("calculateFinalPrice", () => {
  it("должна возвращать базовую цену + сервисный сбор, если basePrice > 0", () =>
{
    expect(calculateFinalPrice(100)).toBe(150);
    expect(calculateFinalPrice(1)).toBe(51);
  });
```

```
  it("должна возвращать только сервисный сбор, если basePrice = 0", () => {
    expect(calculateFinalPrice(0)).toBe(50);
  });

  it("должна возвращать только сервисный сбор, если basePrice < 0", () => {
    expect(calculateFinalPrice(-100)).toBe(50);
  });
});
```

Вначале файле, мы импортировали функции **describe, expect, it** из библиотеки **Vitest**, чтобы выполнять тестирование. Затем импортируется функция **calculateFinalPrice**, которую необходимо проверить. Блок **describe** создаёт группу тестов с общим названием, чтобы логически объединить проверки, относящиеся к одной функции.

1) Функция должна корректно рассчитывать итоговую стоимость тура, добавляя сервисный сбор к базовой цене, если базовая цена больше нуля.
2) Функция должна возвращать сумму, состоящую только из сервисного сбора, если базовая цена равна нулю.
3) Функция должна возвращать сумму, равную сервисному сбору, если базовая цена имеет отрицательное значение.

В этом тесте также создаются входное значение и ожидаемый результат, после чего выполняется сравнение через **expect**. Оба теста выполняют вызов одной и той же функции, но проверяют разные возможные сценарии её работы, чтобы убедиться, что функция ведёт себя правильно в обычной и экстремальной ситуации.

5. **Запуск тестов**. Для запуска всех тестов в проекте используется команда:

```
npm test
# или
npm run test:watch
```

После запуска теста, получаем следующий результат:

```
RERUN src/tests/priceUtils.test.js x3

 ✓ src/tests/priceUtils.test.js (3 tests) 2ms
   ✓ calculateFinalPrice (3)
     ✓ должна возвращать базовую цену + сервисный сбор, если basePrice > 0 1ms
     ✓ должна возвращать только сервисный сбор, если basePrice = 0 0ms
     ✓ должна возвращать только сервисный сбор, если basePrice < 0 0ms

  Test Files  1 passed (1)
       Tests  3 passed (3)
    Start at  18:49:56
    Duration  16ms

PASS Waiting for file changes...
       press h to show help, press q to quit
```

Vitest сообщает, что он нашёл один файл с тестами и успешно выполнил 3 теста внутри него. В строке **src/tests/priceUtils.test.js** видно, что тесты находятся именно в этом файле.

Под названием группы **calculateFinalPrice** (основная логика) отображаются оба теста, и зелёные галочки показывают, что они прошли успешно. Первой строкой отмечен тест, который проверяет добавление сервисного сбора к базовой цене, и рядом написано время выполнения — одна миллисекунда. Второй тест проверяет ситуацию с нулевой ценой, и он тоже прошёл без ошибок.

Ниже Vitest пишет **Test Files 1 passed (1)** и **Tests 3 passed (3)**, что означает, что все найденные тестовые файлы и все тесты внутри них завершились успешно. Время запуска тестов указано отдельно, и оно показывает общую скорость выполнения. В конце Vitest переходит в режим ожидания изменений, что говорит о том, что тестовый раннер запущен в режиме watch и автоматически перезапустит тесты, если ты изменишь файлы.

Главное, на что нужно обращать внимание, - это зелёные галочки, отсутствие ошибок, количество прошедших тестов и то, что все тесты совпадают с ожидаемым количеством. Если всё зелёное и количество **passed** совпадает с общим количеством тестов, значит логика работает корректно.

Основы синтаксиса Jest: Describe, Test, Expect

Прежде чем мы начнем рендерить React-компоненты, нам нужно выучить «грамматику» тестов. Хорошая новость: синтаксис Jest (и Vitest) очень прост и читается почти как английские предложения.

Сейчас мы не будем трогать React. Мы будем тестировать обычный JavaScript. Это поможет вам понять суть, не отвлекаясь на DOM и HTML.

Любой тест состоит из трех главных элементов:

1) **Оболочка** (**test** или **it**): само описание того, что мы проверяем.

2) **Утверждение** (**expect**): ожидание того, какой результат мы должны получить.

3) **Матчеры** (**Matchers**): способ сравнения результата (равно, содержит, больше чем).

И часто мы используем группировку **describe**, чтобы наводить порядок.

Анатомия теста: test и it

Это минимальная единица тестирования. Функция принимает два аргумента:

- Название теста (строка).
- Функцию, внутри которой происходит проверка.

Для всех последующих примеров тестов, определим в папке **test**, файл **general.test.js**.

Рассмотрим пример, изменим файл **general.test.js**:

```
import { expect, test } from "vitest";
```

```
test("the sum of 2 + 2 must equal 4", () => {
  const result = 2 + 2;
  expect(result).toBe(4);
});
```

Вначале этого файла, мы импортировали из библиотеки **Vitest** функции **test** и **expect**, которые нужны для описания тестов и проверки результатов. Мы поместили код в файл **general.test.js** внутри папки **tests**, чтобы Vitest автоматически нашёл и выполнил этот тест.

Мы написали тест с названием "the sum of 2 + 2 must equal 4", чтобы описать, что именно проверяется, именно этот текст будет отображаться в описании теста.

Внутри функции теста мы вычислили выражение **2 + 2** и сохранили результат в переменную **result**. Мы использовали **expect(result).toBe(4)** для проверки, что результат действительно равен **4**. Тест успешно проходит, что подтверждает правильность нашего кода и выбранного подхода.

expect() - это функция, которая получает значение, которое мы хотим проверить. Она возвращает объект, на котором можно вызвать методы-проверки (матчеры), такие как toBe, toEqual, toContain и другие. Это начало утверждения. Мы говорим: «Я ожидаю, что это значение...»

toBe() - это один из матчеров, который проверяет, что значения строго равны (использует === под капотом). Это окончание утверждения. Оно говорит: «…должно быть равно этому значению».

Алиас it

Вместо слова **test** часто используют слово **it**. Технически разницы нет, но it позволяет читать название теста как предложение. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("the sum of 2 + 2 must equal 4", () => {
  const result = 2 + 2;
  expect(result).toBe(4);
});
```

Выберите один стиль (test или it) и придерживайтесь его во всем проекте.

Группировка: describe

Когда тестов становится много, их нужно раскладывать по полочкам. **describe** создает «блок» или «категорию» тестов.

Представьте, что в вашем приложении, есть модуль математики. Мы можем сгруппировать тесты так, изменим файл **general.test.js**:

```
import { describe, test, expect } from "vitest";
```

```
describe("Mathematical calculations", () => {
  describe("Addition", () => {
    test("2 + 2 should equal 4", () => {
      const result = 2 + 2;
      expect(result).toBe(4);
    });

    test("10 + 5 should equal 15", () => {
      const result = 10 + 5;
      expect(result).toBe(15);
    });
  });

  describe("Subtraction", () => {
    test("10 - 2 should equal 8", () => {
      const result = 10 - 2;
      expect(result).toBe(8);
    });
  });
});
```

describe позволяет объединять похожие тесты в блоки. Все тесты про сложение - в одном describe. Все тесты про вычитание - в другом. Это делает структуру тестов понятной и упорядоченной.

Если открыть отчёт тестов или лог консоли, то вывод будет выглядеть так:

```
✓ src/tests/general.test.js (3 tests) 2ms
  ✓ Mathematical calculations (3)
    ✓ Addition (2)
      ✓ 2 + 2 should equal 4 1ms
      ✓ 10 + 5 should equal 15 0ms
    ✓ Subtraction (1)
      ✓ 10 - 2 should equal 8 0ms

 Test Files  1 passed (1)
      Tests  3 passed (3)
   Start at  12:08:07
   Duration  12ms
```

Магия expect и Матчеры (Matchers)

expect(значение) - это начало любой проверки. Но само по себе оно ничего не делает. К нему нужно прицепить «матчер» - метод, который скажет, как именно проверять значение.

Самые популярные матчеры:

toBe() - строгое равенство Использует ===. Подходит для чисел, строк и булевых значений. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  expect(2 + 2).toBe(4);
  expect("Wildland").toBe("Wildland");
});
```

toEqual() - глубокое равенство (ОЧЕНЬ ВАЖНО). Новички часто ошибаются здесь. Если вы сравниваете объекты или массивы, toBe не сработает, потому что в JS два разных объекта с одинаковым содержимым не равны друг другу по ссылке. Используйте toEqual для проверки содержимого. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";
const user = { name: "Alex", app: "Wildland" };
const sameUser = { name: "Alex", app: "Wildland" };

it("Example", () => {
  // Ошибка! Ссылки разные
  expect(user).toBe(sameUser);

  // Успех! Содержимое одинаковое
  expect(user).toEqual(sameUser);
});
```

toBeTruthy() и toBeFalsy() - проверяет, приводится ли значение к true или false (как в if). Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  expect(null).toBeFalsy();
  expect("Hello").toBeTruthy();
});
```

toContain() - проверяет наличие элемента в массиве или подстроки в строке. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  const tours = ["Safari", "Mountain", "Sea"];
  expect(tours).toContain("Safari");
});
```

not - отрицание можно инвертировать любую проверку. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";
```

```
it("Example", () => {
  expect(2 + 2).not.toBe(5);
});
```

toBeDefined() / toBeUndefined() - проверяют, определена ли переменная. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

const value = "Apple";

it("Example", () => {
  expect(value).toBeDefined();
  expect(valueSecond).toBeUndefined();
});
```

toBeNull() - проверяет, равно ли значение null. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

let value = null;

it("Example", () => {
  expect(value).toBeNull();
});
```

toBeGreaterThan() / toBeLessThan() / toBeGreaterThanOrEqual() / toBeLessThanOrEqual() - используется для сравнение чисел. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  expect(10).toBeGreaterThan(5);
  expect(5).toBeLessThan(10);
  expect(10).toBeGreaterThanOrEqual(10);
  expect(8).toBeLessThanOrEqual(8);
});
```

toMatch() - проверка строки по регулярному выражению или совпадению. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  expect("Wildland Tour").toMatch(/Tour/);
  expect("Hello World").toMatch("World");
});
```

toHaveLength() - проверяет длину массивов, строк и т.п. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  expect([1, 2, 3]).toHaveLength(3);
  expect("Wildland").toHaveLength(8);
});
```

toHaveProperty() - проверяет наличие свойства в объекте + его значение (опционально). Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  const user = { name: "Alex", app: "Wildland", age: 25 };

  expect(user).toHaveProperty("name");
  expect(user).toHaveProperty("age", 25);
});
```

toBeInstanceOf() - проверка, что объект создан через определённый класс. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  class Tour {}
  const safari = new Tour();

  expect(safari).toBeInstanceOf(Tour);
});
```

toThrow() - проверяет, что функция вызывает ошибку. Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  function boom() {
    throw new Error("Explosion!");
  }

  expect(boom).toThrow();
  expect(boom).toThrow("Explosion!");
  expect(boom).toThrow(/Expl/);
});
```

toContainEqual() - как toContain, но сравнивает объекты по содержимому (не по ссылке!). Изменим файл **general.test.js**:

```
import { expect, it } from "vitest";

it("Example", () => {
  const users = [{ name: "Alex" }, { name: "Ivan" }];

  expect(users).toContainEqual({ name: "Alex" });
});
```

Практика: Тестируем утилиту нашего приложения

Допустим, у нас в проекте есть файл **src/utils.js** с функцией форматирования цены тура:

```
export const formatPrice = (price, currency) => {
  if (!price) return "Free";
  return `${price} ${currency}`;
};
```

В папке **src/test**, определим файл теста **utils.test.js**:

```
import { describe, expect, test } from "vitest";
import { formatPrice } from "../utils";

describe("Функция formatPrice", () => {
  test("должна форматировать цену с валютой", () => {
    const result = formatPrice(100, "USD");
    expect(result).toBe("100 USD");
  });

  test('должна возвращать "Free", если цена 0', () => {
    const result = formatPrice(0, "USD");
    expect(result).toBe("Free");
  });

  test("не должна возвращать undefined", () => {
    const result = formatPrice(500, "EUR");
    expect(result).toBeDefined();
  });
});
```

Этот тест проверяет, что функция formatPrice корректно форматирует цену: добавляет валюту, возвращает "Free" при нулевой стоимости и никогда не выдаёт undefined. Он

содержит три сценария, которые подтверждают ожидаемое поведение функции в разных условиях.

Философия React Testing Library и работа с DOM

Как мы уже кратко упоминали, RTL была создана, чтобы противостоять «тестированию деталей реализации» (как в Enzyme).

Золотое правило RTL:

Чем больше ваши тесты похожи на то, как используется ваше программное обеспечение, тем больше уверенности они могут вам дать.

Что это значит на практике?

Когда реальный пользователь заходит на ваш сайт:

- Он не проверяет component.state.isLoading.
- Он не смотрит на пропсы, которые переданы дочернему компоненту.
- Он ищет на странице текст, кнопку, поле ввода или изображение.

RTL (React Testing Library) - предоставляет нам инструменты, которые позволяют взаимодействовать только с тем, что доступно в DOM (Document Object Model).

Когда вы запускаете тесты React-приложения, браузера не существует. Есть только:

- **Node.js** - среда выполнения JavaScript
- **Jest** или **Vitest** - тестовый раннер
- **jsdom** - поддельный браузер

jsdom создаёт виртуальную веб-страницу в памяти:

- у неё есть document
- у неё есть window
- у неё есть HTML-элементы
- у неё нет реального экрана, мыши и клавиатуры

Что именно тестирует RTL?

- RTL не тестирует React.
- RTL не тестирует состояние компонентов.
- RTL не тестирует пропсы напрямую.

RTL тестирует результат работы приложения, а именно:

- что пользователь может увидеть
- что пользователь может найти
- что пользователь может нажать
- что пользователь может ввести

То есть исключительно DOM. Если чего-то нет в DOM - для RTL этого не существует.

Render - момент «оживления» приложения

Пока компонент не отрендерен, его не существует. **render**:

- создаёт виртуальную HTML-страницу
- помещает туда твой React-компонент
- делает его доступным для поиска и взаимодействия

После render:

- DOM существует
- элементы существуют
- пользователь может с ними взаимодействовать

В тесте есть один источник правды - **DOM**. Если:

- текст появился → значит приложение работает
- кнопка исчезла → значит пользователь больше не может нажать
- элемент заблокирован → значит пользователь не может взаимодействовать

Не важно:

- какой state
- какой reducer
- какой эффект сработал

RTL смотрит на **DOM** глазами пользователя, а не разработчика. Поэтому он ищет элементы так, как их ищет человек:

- по видимому тексту
- по подписи поля
- по назначению элемента (кнопка, поле, ссылка)
- по альтернативному тексту изображения

RTL намеренно запрещает:

- заглядывать внутрь компонента
- проверять внутренние переменные
- тестировать частную реализацию
- завязываться на структуру JSX

React Testing Library - это пользователь, который смотрит на DOM и больше ничего не знает.

Тестирование приложения через RTL

React Testing Library (RTL) – это набор утилит для тестирования компонентов React, который фокусируется на тестировании с точки зрения пользователя. Его главный принцип: "Чем больше ваши тесты похожи на то, как используется ваше ПО, тем больше уверенности они могут вам дать".

Создадим новый проект на React:

https://docs.google.com/document/d/1mHXNR8t6La1YKs_eG7E7O-bK5lkSQleV/edit

В проект добавим 3 библотеки: **Vitest**, **Jest** и **RTL**.

Выполним установку библиотеки **Vitest**, в **Terminal** введем команду

npm install -D vitest

vitest - это тестовый раннер. Он отвечает за: запуск тестов (describe, it, test), проверки (expect), отчёты об ошибках, мокинг (vi.fn, vi.mock), управление окружением тестов.

Выполним установку библиотеки **@testing-library/react**, в **Terminal** введем команду

npm install -D @testing-library/react @testing-library/jest-dom

@testing-library/react - это инструмент для тестирования React UI. Он отвечает за: рендер React-компонентов (render), работу с виртуальным DOM, поиск элементов (screen.getByRole, getByText), имитацию пользовательского поведения.

@testing-library/jest-dom - это расширение для удобных DOM-матчеров в expect, таких как:

- toBeInTheDocument() - проверяет, что элемент есть в DOM
- toHaveTextContent() - проверяет текстовое содержимое элемента toBeDisabled(), toHaveAttribute()
- и другие

Чтобы запускать тесты одной командой, необходимо добавить скрипт в ваш файл **package.json**:

```
// package.json
"scripts": {
    "test": "vitest"
  }
```

Создание компонента для тестирования

В папке **src/components**, создадим простой компонент **Button.jsx**, который отображает текст и вызывает функцию при нажатии:

```
import React from "react";

function Button({ label, onClick }) {
  return <button onClick={onClick}>{label}</button>;
}
```

```
export default Button;
```

Создание тестового файла

Создадим файл **Button.test.jsx** в той же директории где и сам компонент. Проверим, что кнопка корректно отображается с заданным текстом:

```
import { render, screen } from "@testing-library/react";
import { describe, it, expect } from "vitest";
import Button from "./Button";

describe("<Button />", () => {
  it("Отображает кнопку с правильным текстом", () => {
    // Рендерим компонент
    render(<Button label="Click me!" onClick={() => {}} />);

    // Получаем элемент кнопки
    const buttonElement = screen.getByRole("button");

    // Проверяем, что элемент существует
    expect(buttonElement).not.toBeNull();

    // Проверяем текст кнопки
    expect(buttonElement.textContent).toBe("Click me!");
  });
});
```

При написании тестов, всегда следуйте шаблону **AAA** (Arrange, Act, Assert):

- **Arrange** (Подготовка): рендеринг компонента, настройка моков.
- **Act** (Действие): симуляция пользовательского взаимодействия (клик, ввод текста).
- **Assert** (Проверка): проверка ожидаемого результата.

Сначала импортируются функции **render** и **screen** из библиотеки **@testing-library/react**, чтобы рендерить компонент и получать доступ к его элементам в виртуальном DOM.

Также импортируются функции **describe**, **it** и **expect** из **vitest**, которые используются для группировки тестов, описания отдельных сценариев и проверки условий.

Импортируется сам компонент **Button**, который будет тестироваться. Функция **describe** создаёт блок тестов с названием <Button />, объединяя все связанные проверки. Внутри этого блока функция **it** описывает конкретный тестовый случай - проверку того, что кнопка отображается с правильным текстом.

Сначала выполняется рендер компонента **Button** с пропсом **label="Click me!"** и пустым обработчиком **onClick**.

Затем через **screen.getByRole("button")** получаем элемент кнопки из виртуального DOM. С помощью **expect(buttonElement).not.toBeNull()** проверяется, что кнопка действительно существует на странице.

Далее **expect(buttonElement.textContent).toBe("Click me!")** проверяет, что текст внутри кнопки соответствует строке **"Click me!"**.

Таким образом, тест гарантирует, что компонент корректно рендерится и отображает правильный текст кнопки.

Запустим и проверим работу теста. В **Terminal**, введем команду:

npm test

В результате выполнения, получим следующую ошибку или **другую подобную**:

```
FAIL src/components/Button.test.jsx > <Button /> > Отображает кнопку с правильным текстом
TypeError: (0 , __vite_ssr_import_2__.render) is not a function
 > src/components/Button.test.jsx:7:5
     5|   it("Отображает кнопку с правильным текстом", () => {
     6|     // 1. Arrange (Подготовка)
     7|     render(<Button label="Click me!" onClick={() => {}} />);
      |     ^
     8|
     9|     // 2. Act (Действие)
```

Эта ошибка означает, что функция render импортировалась не из браузерной версии **@testing-library/react**, а из SSR-окружения, где она не определена.

По умолчанию Vitest может запускать тесты в среде node, а не в jsdom, из-за чего React Testing Library работает некорректно. В результате render существует как импорт, но на самом деле не является функцией.

Чтобы исправить проблему, нужно явно указать браузерное окружение для тестов. Для этого, изменим файл **vite.config.js**, следующим образом:

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";

// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    environment: "jsdom",
  },
});
```

После этого Vitest будет использовать **jsdom**, и **@testing-library/react** сможет корректно рендерить компоненты.

Попробуем запустить тест еще раз:

npm test

Если возникнет следующая ошибка:

```
MISSING DEPENDENCY  Cannot find dependency 'jsdom'

? Do you want to install jsdom? » (y/N)
```

Соглашаемся на установку – **y**. Вводим команду для запуска теста, еще раз:

npm test

```
RERUN  src/components/Button.test.jsx x1

 ✓ src/components/Button.test.jsx (1 test) 98ms
   ✓ <Button /> (1)
     ✓ Отображает кнопку с правильным текстом 97ms

 Test Files  1 passed (1)
      Tests  1 passed (1)
   Start at  12:27:18
   Duration  221ms

 PASS  Waiting for file changes...
       press h to show help, press q to quit
```

Также важный момент. **Файл теста**, должен **иметь расширение .jsx**, если он тестирует компонент вместе с **JSX кодом**.

О поиске элементов (Queries):

screen - используется как унифицированный способ доступа к элементам, отрендеренным в виртуальном DOM:

1) Когда мы вызываем render(<Button ... />), React-компонент рендерится в изолированное виртуальное DOM-дерево, созданное Testing Library.

2) screen предоставляет глобальный доступ к этому дереву без необходимости сохранять возвращаемый объект от render.

3) Через screen можно искать элементы по разным критериям.

RTL предоставляет несколько видов запросов для поиска элементов в DOM. Приоритет поиска должен быть таким, как если бы вы были пользователем с ограниченными возможностями:

- **ByRole**: самый предпочтительный. Ищет по ARIA-роли и (опционально) по доступному имени (name).
- **ByLabelText**: для элементов форм, привязанных к метке (<label>). **ByPlaceholderText**: По тексту заполнителя.
- **ByText**: по видимому текстовому содержимому.
- **ByDisplayValue**: по текущему значению элемента формы.
- **ByTestId**: самый низкий приоритет, используется как запасной вариант (например, для элементов, которые не видны пользователю).

К примеру:

```
    // Получаем элемент кнопки
    const buttonElement = screen.getByRole("button");
    const buttonElement = screen.getByText("Click me!");
```

Каждый запрос имеет 3 вариации:

get*: возвращает элемент или выбрасывает ошибку, если элемент не найден (используется для синхронных проверок, когда элемент должен быть на странице).

К примеру, изменим файл **Button.test.jsx**:

```
import { render, screen } from "@testing-library/react";
import { describe, it, expect } from "vitest";
import Button from "./Button";

describe("<Button />", () => {
  it("Отображает кнопку с правильным текстом", () => {
    // Рендерим компонент
    render(<Button label="Click me!" />);

    // Найдёт кнопку сразу или выбросит ошибку
    const buttonElement = screen.getByRole("button");
    expect(buttonElement.textContent).toBe("Click me!");
  });
});
```

query*: возвращает элемент или null, если элемент не найден (используется для проверки того, что элемент отсутствует на странице).

К примеру, изменим файл **Button.test.jsx**:

```
import { render, screen } from "@testing-library/react";
import { describe, it, expect } from "vitest";
import Button from "./Button";

describe("<Button />", () => {
  it("Попытка найти кнопку, которой нет", () => {
    render(<div>Hello</div>);

    // Попытка найти кнопку, которой нет
    const buttonElement = screen.queryByRole("button");
    expect(buttonElement).toBeNull(); // проверяем, что кнопки нет
  });
});
```

find*: возвращает Promise, который разрешается, когда элемент найден, или отклоняется по таймауту (используется для асинхронных проверок, например, после API-запроса).

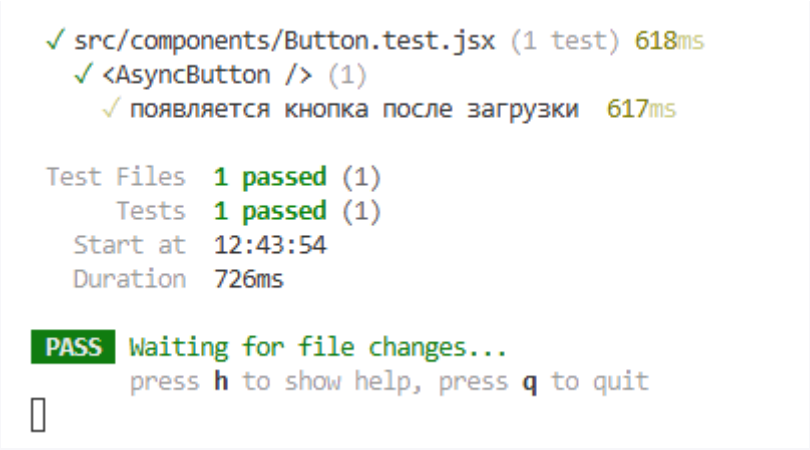
К примеру, изменим файл **Button.test.jsx**:

```
import { render, screen } from "@testing-library/react";
import { describe, it, expect } from "vitest";
import AsyncButton from "./AsyncButton";

describe("<AsyncButton />", () => {
  it("появляется кнопка после загрузки", async () => {
    render(<AsyncButton />);

    // findByRole ждёт появления кнопки (по умолчанию до 1000ms)
    const buttonElement = await screen.findByRole("button");

    expect(buttonElement.textContent).toBe("Loaded!");
  });
});
```

В **src/components**, определим компонент **AsyncButton.jsx**:

```
import { useState, useEffect } from "react";

export default function AsyncButton() {
  const [loaded, setLoaded] = useState(false);

  useEffect(() => {
    const timer = setTimeout(() => {
      setLoaded(true);
    }, 500); // кнопка появится через 0.5 секунды

    return () => clearTimeout(timer);
  }, []);

  if (!loaded) return null;

  return <button>Loaded!</button>;
}
```

Если кнопка не появится в течение таймаута, Promise будет отклонён и тест упадёт:



Проверка взаимодействия

Давайте теперь напишем тем, который проверяет, что при клике на кнопку вызывается переданная функция **onClick**. Для написания этого теста, нам потребуется дополнительная библиотека - **@testing-library/user-event**. Используя **Terminal**, выполним ее установку:

npm install -D @testing-library/user-event

Данная библиотека нужна, чтобы тесты имитировали реальное поведение пользователя, а не просто «дергали события». К примеру, у нее есть объект – **userEvent**, который позволяет имитировать события пользователя. Вызвав такой код:

```
await user.click(button);
```

Что происходит на самом деле:

- pointerdown
- mousedown
- фокус на элементе
- pointerup
- mouseup
- click

Это то, что реально делает пользователь.

Изменим файл **Button.test.jsx**:

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import { describe, it, expect, vi } from "vitest";
import Button from "./Button";

describe("<Button />", () => {
  it("вызывает функцию onClick при клике", async () => {
    // 1. Arrange (Подготовка)
    const user = userEvent.setup();
```

```
    const mockOnClick = vi.fn(); // В Vitest используем vi.fn()
    render(<Button label="Save" onClick={mockOnClick} />);

    const buttonElement = screen.getByRole("button", { name: /Save/i });

    // 2. Act (Действие)
    await user.click(buttonElement);

    // 3. Assert (Проверка)
    expect(mockOnClick).toHaveBeenCalledTimes(1);
  });
});
```

Этот код импортирует функции **render** и **screen** из библиотеки **@testing-library/react** для рендеринга React-компонентов и поиска элементов в виртуальном DOM. Этот код импортирует **userEvent** из @testing-library/user-event для имитации реальных действий пользователя, таких как клики мышью.

Этот код импортирует функции **describe, it, expect и vi** из vitest для описания тестов, их выполнения, проверки ожиданий и создания мок-функций. Этот код импортирует компонент **Button** из локального файла для тестирования его поведения.

Этот код с помощью **describe** группирует тесты, относящиеся к компоненту **<Button />**. Этот код с помощью **it** описывает один конкретный тест, который проверяет вызов функции **onClick** при клике по кнопке.

Этот тест объявлен асинхронным, потому что действия пользователя, выполняемые через **userEvent**, возвращают промисы.

Этот код создаёт объект **user** через **userEvent.setup()** для управления симулируемыми пользовательскими действиями. Этот код создаёт мок-функцию **mockOnClick** с помощью **vi.fn()** для отслеживания количества её вызовов.

Запустим тест **npm test**:

```
 DEV  v4.0.15 D:/React/testapp

 ✓ src/components/Button.test.jsx (1 test) 116ms
   ✓ <Button /> (1)
     ✓ вызывает функцию onClick при клике 115ms

 Test Files  1 passed (1)
      Tests  1 passed (1)
   Start at  16:44:07
   Duration  808ms (transform 31ms, setup 0ms, import 166ms, tests 116ms, environment 416ms)

 PASS  Waiting for file changes...
       press h to show help, press q to quit
```

Этот код рендерит компонент **Button** с текстом **Save** и передаёт мок-функцию в проп **onClick**. Этот код находит кнопку в DOM по её роли **button** и текстовому имени Save, используя **screen.getByRole**. Этот способ поиска имитирует то, как элемент видит пользователь и технологии доступности.

Этот код выполняет клик по кнопке с помощью **await user.click(buttonElement)**, что симулирует реальное поведение пользователя в браузере. Этот клик вызывает цепочку событий, включая фокус и событие **click**.

Этот код проверяет, что функция **mockOnClick** была вызвана ровно один раз с помощью **expect**. Этот тест подтверждает, что компонент **Button** корректно реагирует на пользовательский клик.

Тестирование компонента с внутренним состоянием

Создадим компонент **Counter**, который имеет кнопку для увеличения значения и отображает текущий счетчик. В папку **src/components**, создадим файл **Counter.jsx**, со следующим содержимым:

```
import React, { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      {/* Используем data-testid как запасной вариант для поиска */}
      <p data-testid="count-value">Current score: {count}</p>
      <button onClick={() => setCount((prev) => prev + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

Рядом с компонентом, создадим файл теста **Counter.test.jsx**:

```
import { render, screen } from "@testing-library/react";
import userEvent from "@testing-library/user-event";
import Counter from "./Counter";
import { describe, expect, it } from "vitest";

describe("<Counter />", () => {
  it("счетчик корректно увеличивается после клика", async () => {
    // 1. Arrange (Подготовка)
    render(<Counter />);

    // Получаем элемент, отображающий счетчик, по data-testid
    const countDisplay = screen.getByTestId("count-value");

    // Получаем кнопку для увеличения
    const incrementButton = screen.getByRole("button", { name: /Increment/i });

    // Начальное состояние
    expect(countDisplay.textContent).toContain("Current score: 0");

    // 2. Act (Действие)
```

```
    // Кликаем один раз
    await userEvent.click(incrementButton);

    // Кликаем второй раз
    await userEvent.click(incrementButton);

    // 3. Assert (Проверка)
    // Проверяем, что счетчик обновился в DOM
    expect(countDisplay.textContent).toContain("Current score: 2");
  });
});
```

Функция **describe** группирует тесты, относящиеся к компоненту <Counter />, и задаёт читаемое описание этого набора. Функция **it** объявляет конкретный тестовый сценарий и описывает ожидаемое поведение счётчика после кликов. Асинхронная функция в it используется потому, что действия пользователя и обновления интерфейса происходят не мгновенно.

В шаге **Arrange** компонент **Counter** рендерится в тестовой среде с помощью функции render. Переменная **countDisplay** получает ссылку на DOM-элемент, который отображает текущее значение счётчика и найден по атрибуту **data-testid**. Переменная **incrementButton** получает ссылку на кнопку увеличения, найденную по роли **button** и доступному имени, содержащему слово **Increment**.

Первая проверка с **expect** убеждается, что в начальном состоянии текст счётчика содержит значение **Current score: 0**. В шаге **Act** выполняются действия пользователя, имитирующие два последовательных клика по кнопке увеличения. Каждый вызов **userEvent.click** ожидается с помощью **await**, чтобы дождаться завершения обновления интерфейса.

В шаге Assert выполняется финальная проверка, что текст счётчика в DOM изменился и теперь содержит значение **Current score: 2**. Таким образом тест подтверждает, что компонент корректно реагирует на пользовательские клики и обновляет своё состояние.

Запустим тест **npm test**:

```
RERUN src/components/Counter.test.jsx x8

✓ src/components/Counter.test.jsx (1 test) 213ms
  ✓ <Counter /> (1)
    ✓ счетчик корректно увеличивается после клика 212ms

Test Files  1 passed (1)
     Tests  1 passed (1)
  Start at  17:09:37
  Duration  360ms

PASS Waiting for file changes...
     press h to show help, press q to quit
```

Тестирование форм и ввода пользователя

Создадим простую форму для ввода имени и отображения приветствия. В папке **src/components**, создадим компонент **GreetingForm.jsx**:

```jsx
// src/components/GreetingForm.jsx
import { useState } from "react";

function GreetingForm() {
  // Состояние для хранения введённого имени
  const [name, setName] = useState("");

  // Состояние для хранения текста приветствия
  const [greeting, setGreeting] = useState("");

  // Обработчик отправки формы
  const handleSubmit = (event) => {
    // Предотвращаем перезагрузку страницы
    event.preventDefault();

    // Формируем текст приветствия на основе введённого имени
    setGreeting(`Hello, ${name}!`);
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* label связывается с input через htmlFor и id */}
      {/* Это позволяет искать input через getByLabelText */}
      <label htmlFor="name-input">Name:</label>

      <input
        id="name-input"
        type="text"
        value={name}
        // Обновляем состояние при каждом вводе символа
        onChange={(event) => setName(event.target.value)}
      />

      {/* Кнопка отправки формы */}
      <button type="submit">Send</button>

      {/* Приветствие отображается только после отправки формы */}
      {greeting && <h3 data-testid="greeting-message">{greeting}</h3>}
    </form>
  );
}

export default GreetingForm;
```

В том же месте, определим файл теста **GreetingForm.test.jsx**:

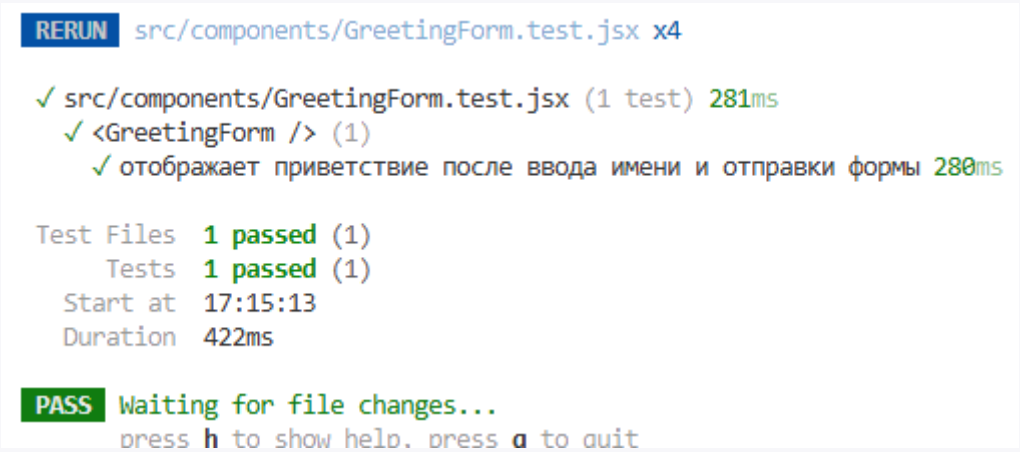
```jsx
import { render, screen } from "@testing-library/react";
```

```
import userEvent from "@testing-library/user-event";
import { describe, it, expect } from "vitest";
import GreetingForm from "./GreetingForm";

describe("<GreetingForm />", () => {
  it("отображает приветствие после ввода имени и отправки формы", async () => {
    // ====================
    // 1. Arrange (Подготовка)
    // ====================

    // Рендерим компонент формы в виртуальный DOM
    render(<GreetingForm />);

    // Находим поле ввода по связанному label
    // Это имитирует поведение реального пользователя
    const nameInput = screen.getByLabelText(/Name:/i);

    // Находим кнопку отправки по роли и тексту
    const submitButton = screen.getByRole("button", {
      name: /Send/i,
    });

    // ====================
    // 2. Act (Действие)
    // ====================

    // Имитируем ввод текста пользователем
    // userEvent.type вводит символы так же, как реальный пользователь
    await userEvent.type(nameInput, "Alex");

    // Имитируем клик по кнопке отправки формы
    await userEvent.click(submitButton);

    // ====================
    // 3. Assert (Проверка)
    // ====================

    // Находим элемент с приветствием по data-testid
    const greetingMessage = screen.getByTestId("greeting-message");

    // Проверяем, что приветствие появилось в DOM
    expect(greetingMessage).not.toBeNull();

    // Проверяем, что текст приветствия сформирован корректно
    expect(greetingMessage.textContent).toContain("Hello, Alex!");
  });
});
```

Этот тест рендерит компонент **<GreetingForm />** в виртуальном DOM, имитирует ввод имени пользователем и нажатие кнопки отправки, а затем проверяет, что после этого

на странице появляется корректное приветствие. Он использует screen для поиска элементов, **userEvent** для симуляции действий пользователя и expect для проверки правильности отображаемого текста.

Запустим тест **npm test**:



Тестирование асинхронных операций (Mocking API)

Это самый частый реальный сценарий. Мы должны проверить:

1) Отображается ли состояние загрузки (Loading...).

2) Как только данные получены (успешный ответ), исчезает ли состояние загрузки и отображаются ли данные.

3) Если произошла ошибка, отображается ли сообщение об ошибке.

Для этого нам нужно имитировать (**mock**) сетевые запросы, чтобы тест был быстрым и независимым от реального API.

В папке **src/components**, создадим компонент **UserList.jsx**:

```
import { useState, useEffect } from "react";

function UserList() {
  // Состояние для хранения списка пользователей
  const [users, setUsers] = useState([]);

  // Состояние загрузки
  const [loading, setLoading] = useState(true);

  // Загружаем данные при первом рендере
  useEffect(() => {
    fetch("/api/users")
      .then((res) => res.json())
      .then((data) => {
        setUsers(data); // Обновляем список пользователей
        setLoading(false); // Снимаем индикатор загрузки
      });
  }, []);
```

```
  if (loading) {
    // Пока данные загружаются, показываем сообщение
    return <div>User loading...</div>;
  }

  // Когда данные загружены, отображаем список
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default UserList;
```

useState управляет состоянием компонента (данные и индикатор загрузки). **useEffect** выполняет асинхронный запрос при монтировании. Пока данные загружаются, компонент возвращает сообщение «Loading…». После получения данных отображается список пользователей.

В том же месте, определим файл теста **UserList.test.jsx**:

```
import { render, screen } from "@testing-library/react";
import { describe, it, beforeEach, afterEach, expect, vi } from "vitest";
import UserList from "./UserList";

describe("<UserList />", () => {
  // Мок данных для теста
  const mockUsers = [
    { id: 1, name: "Alice" },
    { id: 2, name: "Bob" },
  ];

  // Подменяем глобальный fetch перед каждым тестом
  beforeEach(() => {
    global.fetch = vi.fn(() =>
      Promise.resolve({
        json: () => Promise.resolve(mockUsers),
        ok: true,
      })
    );
  });

  // Чистим мок после каждого теста
  afterEach(() => {
    global.fetch = undefined;
  });
```

```
  it("отображает список пользователей после загрузки данных", async () => {
    // =====================
    // 1. Arrange (Подготовка)
    // =====================

    render(<UserList />);

    // Проверяем, что сообщение о загрузке отображается сразу
    const loadingText = screen.getByText("User loading...");
    expect(loadingText.textContent).toContain("User loading...");

    // =====================
    // 2. Act (Действие)
    // =====================

    // Ждем появления пользователей в DOM
    const aliceElement = await screen.findByText("Alice");
    const bobElement = await screen.findByText("Bob");

    // =====================
    // 3. Assert (Проверка)
    // =====================

    // Проверяем, что элементы с именами пользователей существуют
    expect(aliceElement.textContent).toContain("Alice");
    expect(bobElement.textContent).toContain("Bob");

    // Проверяем, что сообщение о загрузке больше не существует
    const loadingGone = screen.queryByText("User loading...");
    expect(loadingGone).toBeNull();

    // Проверяем, что fetch был вызван один раз с правильным URL
    expect(global.fetch.mock.calls.length).toBe(1);
    expect(global.fetch.mock.calls[0][0]).toBe("/api/users");
  });
});
```

Этот код описывает тест для компонента **<UserList />** с использованием **Vitest** и **React Testing Library**. В начале задаётся массив **mockUsers**, который содержит фиктивные данные пользователей для использования в тесте.

Перед каждым тестом с помощью **beforeEach** глобальная функция **fetch** подменяется на мок-функцию **vi.fn**, которая возвращает промис с заранее определёнными данными и статусом успешного ответа.

После каждого теста с помощью **afterEach** глобальная функция **fetch** очищается, чтобы не повлиять на последующие тесты. Тест с описанием "отображает список пользователей после загрузки данных" сначала рендерит компонент **<UserList />** в виртуальный DOM с помощью **render**.

Сразу после рендера проверяется, что на странице отображается текст загрузки "User loading..." с помощью **getByText** и проверки **textContent**. Далее с помощью

findByText асинхронно ожидается появление элементов с именами пользователей "Alice" и "Bob", что имитирует получение данных с сервера.

После этого проверяется, что элементы с именами пользователей действительно присутствуют в DOM, используя **textContent** и стандартные проверки. Затем проверяется, что текст загрузки исчез из DOM с помощью **queryByText**, который возвращает null, если элемент больше не существует.

В конце теста проверяется, что мок **fetch** был вызван ровно один раз и с правильным URL, что подтверждает корректное выполнение запроса к API.

Таким образом, тест проверяет последовательность действий: отображение состояния загрузки, появление данных после получения ответа и исчезновение сообщения о загрузке, а также правильность вызова сетевого запроса.

Тестирование асинхронности и взаимодействие с API

Давайте создадим несколько простых React-компонентов, которые работают с внешним API пользователей. Покроем основные сценарии тестирования асинхронности: загрузка, успех, ошибка, отмена запроса, гонки запросов и отправка форм.

Напишем тесты только с помощью **React Testing Library** (RTL) и **Vitest** (без MSW, без Cypress).

Для примеров будем использовать публичный фейковый API JSONPlaceholder:

https://jsonplaceholder.typicode.com/

Создадим новый проект на React:

https://docs.google.com/document/d/1mHXNR8t6La1YKs_eG7E7O-bK5lkSQleV/edit

В проект добавим 3 библотеки: **Vitest**, **Jest** и **TLR**.

npm install -D vitest @testing-library/react @testing-library/jest-dom jsdom

Чтобы запускать тесты одной командой, необходимо добавить скрипт в ваш файл **package.json**:

```
// package.json
"scripts": {
    "test": "vitest"
  }
```

В корне проекта, создадим **vitest.setup.js**:

```
import "@testing-library/jest-dom";
```

по умолчанию Vitest поддерживает только базовые matchers (toBe, toEqual и т.п.). Методы вроде toHaveTextContent, toBeInTheDocument, toHaveAttribute из @testing-library/jest-dom без подключения не работают и вызывают ошибки.

То есть файл нужен для глобальной подготовки тестового окружения. Без него ваши тесты с toHaveTextContent или toBeInTheDocument не будут работать.

Укажем браузерное окружение для тестов. Для этого, изменим файл **vite.config.js**, следующим образом:

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";

// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    environment: "jsdom",
    globals: true,
    setupFiles: "./vitest.setup.js",
  },
});
```

В этом файле мы настроили Vite и Vitest для проекта на React:

plugins: [react()] - подключаем плагин React, чтобы Vite мог обрабатывать JSX/TSX и реактовские фичи.

test: { ... } - конфигурация Vitest: environment: "jsdom" - используем DOM-окружение, чтобы тесты React могли работать с document и window. globals: true - позволяет писать тесты без импорта describe, it, expect в каждом файле.

setupFiles: **"./vitest.setup.js"** - указываем файл, который выполняется перед всеми тестами, чтобы подключить, например, matchers из @testing-library/jest-dom.

После этой настройки можно писать тесты React с RTL и использовать расширенные expect-методы вроде toBeInTheDocument и toHaveTextContent.

Компоненты

Мы сделаем три компонента:

UserList - загружает список пользователей и отображает их.

UserDetails - загружает детальную информацию по одному пользователю (по id).

CreateUserForm - простая форма, которая POST-ит данные на сервер (демонстрация POST-запросов).

В папке **src/components** определим компонент **UserList.jsx**:

```
import { useState, useEffect } from "react";
```

```
export function UserList({
  apiBase = "https://jsonplaceholder.typicode.com",
  fetchImpl = fetch,
  onUserSelect,
}) {
  const [users, setUsers] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  useEffect(() => {
    let cancelled = false;
    setLoading(true);

    fetchImpl(`${apiBase}/users`)
      .then((res) => {
        if (!res.ok) throw new Error(`HTTP ${res.status}`);
        return res.json();
      })
      .then((data) => {
        if (!cancelled) setUsers(data);
      })
      .catch((err) => {
        if (!cancelled) setError(err);
      })
      .finally(() => {
        if (!cancelled) setLoading(false);
      });

    return () => {
      cancelled = true;
    };
  }, [apiBase, fetchImpl]);

  if (loading) return <div role="status">Loading...</div>;
  if (error) return <div role="alert">Error: {error.message}</div>;
  if (!users) return null;

  return (
    <ul>
      {users.map((u) => (
        <li
          key={u.id}
          style={{ cursor: "pointer", marginBottom: "0.5rem" }}
          onClick={() => onUserSelect?.(u.id)}
        >
          {u.name} — {u.email}
        </li>
      ))}
    </ul>
  );
```

```
}
```

В папке **src/components** определим компонент **UserDetails.jsx**:

```
import React, { useEffect, useState } from "react";

export function UserDetails({
  id,
  apiBase = "https://jsonplaceholder.typicode.com",
  fetchImpl = fetch,
}) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  useEffect(() => {
    if (!id) return;
    let abort = false;
    setLoading(true);
    fetchImpl(`${apiBase}/users/${id}`)
      .then((res) => {
        if (!res.ok) throw new Error(`HTTP ${res.status}`);
        return res.json();
      })
      .then((data) => {
        if (!abort) setUser(data);
      })
      .catch((err) => {
        if (!abort) setError(err);
      })
      .finally(() => {
        if (!abort) setLoading(false);
      });

    return () => {
      abort = true;
    };
  }, [id, apiBase, fetchImpl]);

  if (loading) return <div role="status">Loading details...</div>;
  if (error) return <div role="alert">Error: {error.message}</div>;
  if (!user) return <div>No user selected</div>;

  return (
    <div>
      <h2>{user.name}</h2>
      <p>{user.email}</p>
      <p>{user.phone}</p>
    </div>
  );
```

```
}
```

В папке **src/components** определим компонент **CreateUserForm.jsx**:

```
import React, { useState } from "react";

export function CreateUserForm({
  apiBase = "https://jsonplaceholder.typicode.com",
  fetchImpl = fetch,
  onSuccess,
}) {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  async function handleSubmit(e) {
    e.preventDefault();
    setError(null);
    if (!name || !email) {
      setError(new Error("name and email required"));
      return;
    }
    setLoading(true);
    try {
      const res = await fetchImpl(`${apiBase}/users`, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ name, email }),
      });
      if (!res.ok) throw new Error(`HTTP ${res.status}`);
      const data = await res.json();
      onSuccess?.(data);
    } catch (err) {
      setError(err);
    } finally {
      setLoading(false);
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        placeholder="Name"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <input
        placeholder="Email"
```

```
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <button type="submit" disabled={loading}>
        Create
      </button>
      {loading && <div role="status">Sending...</div>}
      {error && <div role="alert">{error.message}</div>}
    </form>
  );
}
```

Изменим компонент **App.jsx**:

```
import { useState } from "react";
import { UserList } from "./components/UserList";
import { UserDetails } from "./components/UserDetails";
import { CreateUserForm } from "./components/CreateUserForm";

function App() {
  // state для выбранного пользователя
  const [selectedUserId, setSelectedUserId] = useState(null);

  return (
    <div style={{ display: "flex", gap: "2rem" }}>
      {/* Список пользователей с callback на выбор */}
      <div>
        <h2>Users</h2>
        <UserList onUserSelect={setSelectedUserId} />
      </div>

      {/* Детали выбранного пользователя */}
      <div>
        <h2>User Details</h2>
        <UserDetails id={selectedUserId} />
      </div>

      {/* Форма создания пользователя */}
      <div>
        <h2>Create User</h2>
        <CreateUserForm />
      </div>
    </div>
  );
}

export default App;
```

Запустим приложение и проверим его работу.

Структура тестов - какие сценарии покрываем

Для каждого компонента напишем тесты на:

- Успешную загрузку (статус 200) - проверяем рендер данных.
- Поведение при загрузке (показывается индикатор загрузки).
- Ошибочный ответ (500 или reject) - проверяем отображение ошибки.
- Отмена запроса при размонтировании (нет утечек state updates).
- Гонка запросов (два запроса - последний победил).
- POST-запрос: успешный/ошибочный, валидация формы.

Для тестов создадим папку **/src/tests**.

Вначале создадим общую утилиту для всех тестов. Определим файл **test-utils.js**:

```
import React from "react";
import { render } from "@testing-library/react";

export * from "@testing-library/react";

// Обёртка для рендера компонента с доп. опциями
export function customRender(ui, options) {
  return render(ui, { ...options });
}
```

Этот код создаёт утилиты для тестирования React-компонентов с помощью React Testing Library:

1) Импортируется React, потому что компоненты используют JSX.

2) Импортируется функция render из @testing-library/react, которая отвечает за отрисовку компонентов в тестовом DOM.

3) Экспортируются все функции из **@testing-library/react**, чтобы их можно было использовать напрямую в тестах (screen, fireEvent и т.д.).

4) Создаётся функция **customRender**, которая оборачивает **render** и позволяет передавать дополнительные опции при рендере компонента.

5) Таким образом, можно использовать **customRender** вместо стандартного render, если нужны кастомные настройки, и при этом иметь доступ ко всем стандартным утилитам RTL.

Напишем тесты для **UserList**, определим файл **UserList.test.jsx**:

```
import { describe, it, expect, vi, beforeEach, afterEach } from "vitest";
import { render, screen, waitFor } from "@testing-library/react";
import { UserList } from "../components/UserList";

const sampleUsers = [
  { id: 1, name: "Alice", email: "alice@example.com" },
```

```
  { id: 2, name: "Bob", email: "bob@example.com" },
];

describe("UserList (mocked fetch)", () => {
  beforeEach(() => {
    global.fetch = vi.fn();
  });
  afterEach(() => {
    vi.restoreAllMocks();
  });

  it("renders users after successful fetch", async () => {
    // Мокируем fetch, возвращаем sampleUsers
    global.fetch.mockResolvedValue({ ok: true, json: async () => sampleUsers });

    render(<UserList fetchImpl={global.fetch} />);

    // Проверяем индикатор загрузки
    expect(screen.getByRole("status")).toHaveTextContent(/Loading/i);

    // Ждём появления пользователей
    await waitFor(() => {
      expect(screen.getByText(/Alice/)).toBeInTheDocument();
      expect(screen.getByText(/Bob/)).toBeInTheDocument();
    });
  });

  it("shows error when fetch rejects", async () => {
    global.fetch.mockRejectedValue(new Error("network down"));

    render(<UserList fetchImpl={global.fetch} />);

    await waitFor(() =>
      expect(screen.getByRole("alert")).toHaveTextContent(/network down/)
    );
  });
});
```

Этот код создаёт тесты для компонента UserList с использованием Vitest и React Testing Library:

1) Импортируются функции из Vitest (describe, it, expect, vi, beforeEach, afterEach) для написания тестов, создания моков и организации подготовки/очистки перед тестами.

2) Импортируются функции render, screen и waitFor из RTL для рендеринга компонента и проверки его отображения в тестовом DOM. Импортируется сам компонент **UserList**, который мы хотим протестировать.

3) Определяется массив **sampleUsers** с примерными данными пользователей, который будет возвращаться при мокировании fetch. В блоке describe объединяются тесты, относящиеся к мокированию **fetch** для компонента **UserList**.

4) В **beforeEach** создаётся мок для **global.fetch** с помощью **vi.fn()**, чтобы каждый тест начинался с чистого состояния. В **afterEach** выполняется **vi.restoreAllMocks()**, чтобы очищать моки после каждого теста.

5) Первый тест "renders users after successful fetch" проверяет успешную загрузку пользователей:

- мок fetch возвращает sampleUsers;
- рендерится компонент;
- проверяется индикатор загрузки;
- с помощью waitFor ждём, пока на странице появятся элементы с именами пользователей.

Второй тест "shows error when fetch rejects" проверяет обработку ошибок:

- мок fetch отклоняется с ошибкой "network down";
- рендерится компонент;
- с помощью waitFor проверяем, что на странице отображается сообщение об ошибке.

Таким образом, код покрывает основные сценарии компонента **UserList**: успешная загрузка и ошибка запроса, используя мокирование сетевых вызовов и асинхронное ожидание DOM.

Запустим тест и проверим его работу:

```
RERUN src/tests/UserList.test.jsx x2

 ✓ src/tests/UserList.test.jsx (2 tests) 128ms
   ✓ UserList (mocked fetch) (2)
     ✓ renders users after successful fetch 107ms
     ✓ shows error when fetch rejects 20ms

 Test Files  1 passed (1)
      Tests  2 passed (2)
   Start at  18:41:51
   Duration  250ms

 PASS Waiting for file changes...
      press h to show help, press q to quit
```

Напишем тесты для **CreateUserForm**, в папке **src/tests** определим файл **CreateUserForm.test.jsx**:

```
import { render, screen, fireEvent, waitFor } from "@testing-library/react";
import { vi } from "vitest";
import { CreateUserForm } from "../components/CreateUserForm";

it("validates required fields", () => {
  render(<CreateUserForm fetchImpl={global.fetch} />);

  fireEvent.click(screen.getByRole("button", { name: /create/i }));
```

```
  expect(screen.getByRole("alert")).toHaveTextContent(/required/);
});

it("sends POST and calls onSuccess", async () => {
  const mockRes = { id: 123, name: "Zoe", email: "z@x.com" };
  global.fetch = vi
    .fn()
    .mockResolvedValue({ ok: true, json: async () => mockRes });
  const onSuccess = vi.fn();

  render(<CreateUserForm fetchImpl={global.fetch} onSuccess={onSuccess} />);

  fireEvent.change(screen.getByPlaceholderText(/name/i), {
    target: { value: "Zoe" },
  });
  fireEvent.change(screen.getByPlaceholderText(/email/i), {
    target: { value: "z@x.com" },
  });
  fireEvent.click(screen.getByRole("button", { name: /create/i }));

  await waitFor(() => expect(onSuccess).toHaveBeenCalledWith(mockRes));
});
```

Этот код создаёт тесты для компонента **CreateUserForm** с использованием Vitest и React Testing Library.

Первый тест "validates required fields":

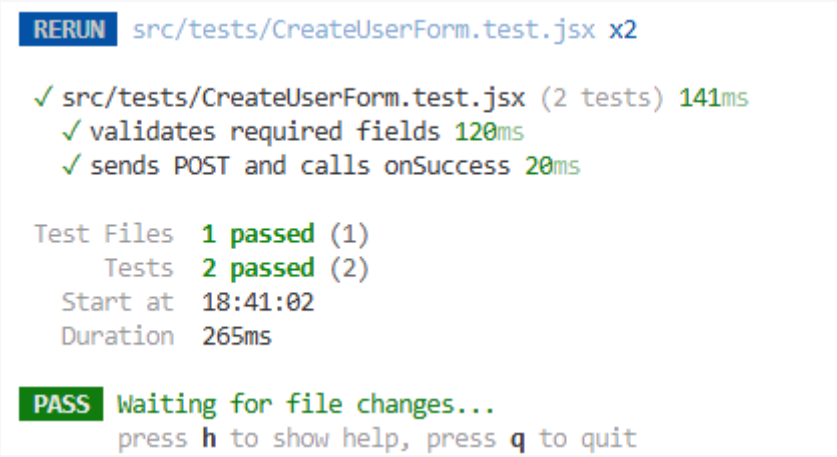
- Рендерится форма без заполненных полей.
- С помощью fireEvent.click имитируется нажатие на кнопку «Create».
- Проверяется, что появляется сообщение об ошибке валидации (role="alert"), содержащее текст «required».
- Этот тест проверяет, что форма корректно реагирует на пустые обязательные поля.

Второй тест "sends POST and calls onSuccess":

- Создаётся мок для fetch, который возвращает успешный ответ с данными нового пользователя mockRes.
- Создаётся шпион onSuccess, чтобы проверить, вызывается ли колбэк после успешного POST-запроса.
- Рендерится компонент с переданными fetchImpl и onSuccess.
- С помощью fireEvent.change заполняются поля формы (name и email).
- Кликается кнопка «Create» для отправки формы.
- Через waitFor проверяется, что onSuccess был вызван с ожидаемыми данными mockRes.
- Этот тест проверяет корректную отправку POST-запроса, обработку ответа и вызов колбэка при успешной операции.

Таким образом, код покрывает валидацию формы и успешную отправку данных на сервер, используя мокирование сетевых вызовов и асинхронное ожидание DOM.

Запустим тест и проверим его работу:



Техники мокирования (Mocking) данных и функций.

Мокирование (Mocking) - это процесс замены реальных, зависимых компонентов, модулей или данных в вашем коде контролируемыми, симулированными версиями. Это критически важный инструмент, особенно в разработке, ориентированной на тестирование (TDD) и юнит-тестировании.

Мок (Mock) - это по сути поддельный объект, который имитирует поведение настоящего объекта, но находится под вашим полным контролем:

- **Цель**: изолировать тестируемый модуль (функцию, компонент) от его зависимостей, чтобы проверить только его собственную логику.
- **Имитация**: моки позволяют вам имитировать ответы сетевых запросов, возвращаемые значения функций, или даже целые модули (например, сторонние библиотеки).

Зачем использовать мокирование?

Использование моков делает тесты:

1) **Быстрыми**: тесты не ждут реальных сетевых ответов или выполнения сложной логики.

2) **Надёжными**: результат теста всегда будет одинаковым, независимо от внешних факторов (например, доступности API, состояния базы данных).

3) **Изолированными**: вы тестируете только один компонент или функцию. Если тест провалился, вы точно знаете, что проблема в тестируемом коде, а не в его зависимостях.

4) **Управляемыми**: вы можете имитировать специфические сценарии, такие как ошибки сети, пустые данные или очень длинные ответы, которые сложно воспроизвести в реальных условиях.

Основные объекты для Мокирования в React

В контексте React и JavaScript чаще всего мокируют следующее:

Объект	Что Мокируем	Инструмент/Техника
Сетевые Запросы	`fetch`, `axios`, `GraphQL` API ответы	`jest-fetch-mock`, `msw` (Mock Service Worker), `Nock`
Функции/ Модули	Функции-помощники, сторонние утилиты, `console.log`	`jest.fn()`, `jest.mock('module-name')`
Хуки (Hooks)	`useContext`, `useSelector` (Redux)	`jest.mock`, создание фиктивных провайдеров
Браузерные API	`localStorage`, `setTimeout`	`jest.spyOn`, глобальное мокирование

В экосистеме React, Jest является стандартом для тестирования и предоставляет мощный набор инструментов для мокирования.

Создадим новый проект на React:

https://docs.google.com/document/d/1mHXNR8t6La1YKs_eG7E7O-bK5lkSQleV/edit

В проект добавим 3 библотеки: **Vitest**, **Jest** и **TLR**.

npm install -D vitest @testing-library/react @testing-library/jest-dom jsdom

Чтобы запускать тесты одной командой, необходимо добавить скрипт в ваш файл **package.json**:

```
// package.json
"scripts": {
    "test": "vitest"
  }
```

В корне проекта, создадим **vitest.setup.js**:

```
import "@testing-library/jest-dom";
```

по умолчанию Vitest поддерживает только базовые matchers (toBe, toEqual и т.п.). Методы вроде toHaveTextContent, toBeInTheDocument, toHaveAttribute из @testing-library/jest-dom без подключения не работают и вызывают ошибки.

То есть файл нужен для глобальной подготовки тестового окружения. Без него ваши тесты с toHaveTextContent или toBeInTheDocument не будут работать.

Укажем браузерное окружение для тестов. Для этого, изменим файл **vite.config.js**, следующим образом:

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";

// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    environment: "jsdom",
    globals: true,
    setupFiles: "./vitest.setup.js",
  },
});
```

В этом файле мы настроили Vite и Vitest для проекта на React.

Так же выполним установку библиотеки:

npm install -D @testing-library/user-event

Эта библиотека необходима чтобы, эмулировать реальные действия пользователя, а не просто «дергать события».

Мокирование функций с jest.fn() (Mock Functions)

Это самый базовый и часто используемый инструмент. jest.fn() создаёт "фиктивную" функцию, которая позволяет вам:

- Проверить, была ли она вызвана.
- Проверить, с какими аргументами она была вызвана.
- Проверить, сколько раз она была вызвана.
- Установить, что она должна возвращать (например, mockReturnValue).

Давайте выполним тестирование компонента-кнопки, который принимает обработчик клика. В папке **src/components**, создадим компонент **Button.jsx**:

```
const Button = ({ onClick, children }) => (
  <button onClick={onClick}>{children}</button>
);

export default Button;
```

В этой же папке, создадим файл теста **Button.test.jsx**:

```
import { render, screen } from "@testing-library/react";
```

```
import userEvent from "@testing-library/user-event";
import Button from "./Button";
import { expect, test, vi } from "vitest";

test("вызывает функцию onClick при клике", async () => {
  // 1. Создаем мок-функцию
  const mockHandleClick = vi.fn();

  render(<Button onClick={mockHandleClick}>Click me!</Button>);
  const button = screen.getByText(/Click me!/i);

  // 2. Имитируем клик пользователя
  await userEvent.click(button);

  // 3. Проверяем, что мок-функция была вызвана
  expect(mockHandleClick).toHaveBeenCalledTimes(1);

  // Опционально: Проверяем, что она была вызвана без аргументов
  expect(mockHandleClick).toHaveBeenCalledWith(expect.any(Object));
});
```

Этот тест проверяет, что компонент кнопки корректно реагирует на действие пользователя. Сначала создаётся мок-функция, которая имитирует обработчик клика и позволяет отследить факт её вызова. Затем компонент кнопки рендерится с этой функцией, после чего тест имитирует реальный клик пользователя по кнопке.

После клика тест проверяет, что переданная функция была вызвана ровно один раз и что при вызове она получила объект события клика. Таким образом, тест подтверждает, что компонент правильно прокидывает обработчик **onClick** и реагирует на пользовательское взаимодействие так, как ожидается.

await userEvent.click(button) - этот метод имитирует реальный клик пользователя по кнопке. Он не просто вызывает обработчик, а последовательно проигрывает все связанные события браузера (фокус, нажатие и отпускание кнопки мыши, клик). Ключевое слово await используется потому, что эти действия происходят асинхронно и тест должен дождаться их завершения, прежде чем продолжать проверки.

expect(mockHandleClick).toHaveBeenCalledTimes(1) - этот метод проверяет, сколько раз была вызвана мок-функция. В данном случае тест утверждает, что обработчик клика должен быть вызван ровно один раз после одного пользовательского клика. Если функция не была вызвана или была вызвана больше одного раза, тест завершится с ошибкой.

expect(mockHandleClick).toHaveBeenCalledWith(expect.any(Object)) - этот метод проверяет, с какими аргументами была вызвана мок-функция. Здесь тест утверждает, что функция была вызвана с каким-либо объектом, не уточняя его точную структуру. Обычно таким объектом является событие клика, которое React автоматически передаёт в обработчик onClick.

Мокирование Модулей с jest.mock() (Не работает, пропускаем)

Если ваш компонент или функция зависит от целого модуля (например, сторонней библиотеки для логирования или утилитарного файла), вы можете мокировать его целиком.

В папке **src**, определим файл **api.js**:

```
export const fetchData = async (id) => {
  const response = await fetch(`/api/items/${id}`);
  return response.json();
};
```

Это реальный код, который мы НЕ хотим выполнять в тесте (он делает fetch).

В папке **src/components**, создадим компонент **ItemDetails.jsx**:

```
import { useEffect, useState } from "react";
import { fetchData } from "../api";

const ItemDetails = ({ id }) => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetchData(id).then(setData);
  }, [id]);

  if (!data) {
    return <div>Loading...</div>;
  }

  return <h1>{data.title}</h1>;
};

export default ItemDetails;
```

Теперь, определим тест с мокированием всего модуля. Мы полностью подменим **./api**, и **fetchData** станут мок-функцией. В этой же папке определим **ItemDetails.test.js**:

```
import { render, screen, waitFor } from "@testing-library/react";
import ItemDetails from "./ItemDetails";

// Импортируем модуль, чтобы управлять его моками
import * as api from "../api";
import { expect, test, vi } from "vitest";

vi.mock("../api");

test("отображает данные после загрузки", async () => {
  // 1. Готовим мок-данные
  const mockData = { id: 1, title: "Mock-Product" };

  // 2. Говорим, что fetchData должен вернуть Promise с данными
  api.fetchData.mockResolvedValue(mockData);

  // 3. Рендерим компонент
```

```
  render(<ItemDetails id={1} />);

  // 4. Проверяем состояние загрузки
  expect(screen.getByText("Loading...")).toBeInTheDocument();

  // 5. Ждём, пока компонент обновится
  await waitFor(() => {
    expect(screen.getByText("Mock-Product")).toBeInTheDocument();
  });

  // 6. Проверяем, что fetchData был вызван правильно
  expect(api.fetchData).toHaveBeenCalledTimes(1);
  expect(api.fetchData).toHaveBeenCalledWith(1);
});
```

Этот код тестирует компонент **ItemDetails**, который загружает данные через функцию **fetchData** из модуля **api**.

Сначала весь модуль api мокируется с помощью **vi.mock**, чтобы реальные HTTP-запросы не выполнялись, и появилась возможность управлять поведением функции **fetchData**.

Далее создаются мок-данные, которые будут возвращены функцией **fetchData**, и устанавливается, что мок-функция возвращает Promise с этими данными.

Компонент **ItemDetails** рендерится с заданным идентификатором, после чего тест проверяет, что на экране сначала отображается сообщение о загрузке.

Затем с помощью **waitFor** тест ждёт, пока компонент обновится после завершения **Promise**, и проверяет, что на странице отображается заголовок из мок-данных. В конце тест проверяет, что мок-функция **fetchData** была вызвана ровно один раз и с правильным аргументом, подтверждая корректное взаимодействие компонента с API.

Мокирование сетевых запросов с MSW (Mock Service Worker)

Для сложных приложений и полноценного E2E (end-to-end) тестирования компонентов, работающих с API, рекомендуется использовать библиотеку **MSW** (Mock Service Worker).

Преимущества MSW (Mock Service Worker):

- Он перехватывает реальные HTTP-запросы на уровне сети (Service Worker в браузере или Node.js), а не просто мокирует fetch или axios.
- Ваш код использует настоящий fetch или axios, что делает тесты более реалистичными.
- Один набор моков можно использовать как для тестов, так и для разработки (имитация API на локальном компьютере).

Рассмотрим пример. Для работы с библиотекой **MSW**, выполним ее установку через **Terminal**:

npm install msw --save-dev

В папке **src/api/** создадим файл **users.js**:

```
export const fetchUsers = async () => {
  const res = await fetch("http://localhost/api/users");
  return res.json();
};
```

В папке **src/components/** создадим компонент, который использует fetchUsers, **UsersList.jsx**:

```
import { useEffect, useState } from "react";
import { fetchUsers } from "../api/users";

const UsersList = () => {
  const [users, setUsers] = useState(null);

  useEffect(() => {
    fetchUsers().then(setUsers);
  }, []);

  if (!users) return <div>Loading...</div>;

  return (
    <ul>
      {users.map((u) => (
        <li key={u.id}>{u.name}</li>
      ))}
    </ul>
  );
};

export default UsersList;
```

Теперь, выполним настройку **MSW**, В папке **src/mocks/** создадим файл, **handlers.js**:

```
import { rest } from "msw";

export const handlers = [
  rest.get("http://localhost/api/users", (req, res, ctx) => {
    return res(
      ctx.status(200),
      ctx.json([
        { id: 1, name: "Alice" },
        { id: 2, name: "Bob" },
      ])
    );
  }),
];
```

Выполним настройку сервера MSW для тестов. В папке **src/** создадим файл, **setupTests.js**:

```
import { setupServer } from "msw/node";
import { handlers } from "./mocks/handlers";
import { afterAll, afterEach, beforeAll } from "vitest";
import "@testing-library/jest-dom";

export const server = setupServer(...handlers);

// Включаем сервер перед всеми тестами
beforeAll(() => server.listen());

// Сбрасываем обработчики после каждого теста
afterEach(() => server.resetHandlers());

// Выключаем сервер после всех тестов
afterAll(() => server.close());
```

Перейдем в файл **vite.config.js** и выполним подключение созданного ранее файла:

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";

// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
  test: {
    environment: "jsdom",
    globals: true,
    setupFiles: "./src/setupTests.js",
  },
});
```

В папке с компонентом **UsersList**, создадим файл тестов **UsersList.test.jsx**:

```
import { render, screen, waitFor } from "@testing-library/react";
import { expect, test } from "vitest";
import UsersList from "./UsersList";

test("отображает список пользователей после загрузки", async () => {
  render(<UsersList />);

  // Проверяем состояние загрузки
  expect(screen.getByText("Loading...")).toBeInTheDocument();

  // Ждём, пока компонент загрузит данные
  await waitFor(() => {
```

```
    expect(screen.getByText("Alice")).toBeInTheDocument();
    expect(screen.getByText("Bob")).toBeInTheDocument();
  });
});
```

Запустим тест и проверим работу приложения.

Лучшие практики (Best Practices) и частые ошибки

Мы научились настраивать окружение, работать с RTL и мокать запросы через MSW. Но наличие тестов само по себе не гарантирует качество. Плохие тесты могут стать кошмаром: они ломаются при каждом рефакторинге, замедляют разработку и не дают уверенности.

Давайте разберем, как писать тесты, которые будут помогать, а не мешать.

1. **Тестируйте поведение, а не детали реализации**. Это главная мантра React Testing Library.

Если вы делаете рефакторинг кода (меняете структуру, оптимизируете хуки), но функционал остается прежним - ваши тесты не должны падать. Если они падают - вы тестируете детали реализации.

2. **Используйте правильные приоритеты селекторов**. Не ищите элементы по **id** или **class** (container, btn-primary). Классы меняются, логика остается. RTL предлагает четкую иерархию (Priority of Queries):

- **getByRole / findByRole**: (лучший выбор) — кнопка, ссылка, чекбокс. Это гарантирует доступность.
- **getByLabelText**: отлично для форм.
- **getByPlaceholderText**: если нет лейбла.
- **getByText**: для поиска неуоинтерактивных элементов (div, span).
- **getByTestId**: (крайняя мера) - используйте data-testid только если ничего другое не подходит.

3. **Избегайте избыточного мокирования**. Не нужно мокать всё подряд. Если вы тестируете компонент «Корзина», пусть он реально рендерит дочерний компонент «Товар». Это **интеграционное тестирование**. Оно дает больше уверенности, чем поверхностные Unit-тесты, где всё замоккано:

- **Мокаем**: запросы к API (MSW), тяжелые браузерные API (Canvas, LocalStorage), время (Timer).
- **Не мокаем**: дочерние компоненты, хуки (если это возможно), внутренние утилиты.

4. **user-event лучше, чем fireEvent**. Вы уже знакомы с обоими, но старайтесь всегда использовать user-event:

- **fireEvent.click(button)** - просто запускает событие клика в DOM.
- **user.click(button)** - симулирует реальное поведение браузера (фокус, hover, нажатие, отпускание). Это находит баги, которые fireEvent пропустит.

5. **Структура проектов с тестами**.

1) Маленькие проекты (малый кодовая база, <10 компонентов). Храните ваши тесты рядом с компонентами.

```
src/
  components/
    Button.jsx
    Button.test.jsx
    Header.jsx
    Header.test.jsx
```

Быстро найти тест для конкретного компонента. Простая структура - не нужно думать, куда класть. Удобно для новых разработчиков: видят компонент и его тест рядом.

2) Средние и большие проекты (10-100+ компонентов). Храните тесты в отдельной папка **tests** или **__tests__**. Централизованная структура тестов. Легко запускать тесты по папкам. Полезно, если есть разные типы тестов: unit, integration, e2e.

3) Тесты разных уровней (Best Practices для больших проектов).

Тип теста	Рекомендуемое место хранения	Примеры
Unit-тесты	Рядом с компонентом или в `__tests__`	`Button.test.jsx`
Integration-тесты	Отдельная папка `tests/integration/`	`tests/integration/Dashboard.test.jsx`
End-to-end	Полностью отдельная папка `e2e/`	`e2e/login.spec.js`
Mock и утилиты	В `tests/utils/`	`tests/utils/mockApi.js`

Пример: Рефакторинг плохого теста в хороший

Представим форму логина, которая показывает сообщение об успехе. **Как не надо тестировать форму**:

```
test("Login form works", () => {
  const wrapper = render(<LoginForm />);

  // Поиск по селекторам реализации (классы, id)
  const input = wrapper.container.querySelector(".login-input");
  const btn = wrapper.container.querySelector("#submit-btn");

  // Использование fireEvent
  fireEvent.change(input, { target: { value: "user" } });
  fireEvent.click(btn);
```

```
  // Проверка внутреннего стейта (невозможно в RTL, но часто пытаются через обходные пути)
  // Или проверка стилей, которые не важны для логики
  expect(btn.className).toBe("btn-active");
});
```

Тест завязан на внутреннюю реализацию компонента, а не на то, что видит пользователь. Если разработчик изменит класс или id, тест сломается, хотя функциональность останется той же.

Использовать getByRole, getByLabelText, getByPlaceholderText из React Testing Library, т.к. они ориентированы на поведение пользователя.

fireEvent - низкоуровневый, и он не имитирует реальное поведение пользователя полностью. Лучше использовать **userEvent** из **testing-library/user-event**, который учитывает фокус, события клавиатуры, задержки и т.д.

RTL ориентирован на тестирование поведения, а не стилей. Стили могут меняться, но функциональность - нет. Проверка внутреннего состояния (например, **useState**) нарушает принцип черного ящика: тест должен проверять что компонент делает, а не как он это делает.

Этот тест ломается при любом рефакторинге классов, id, структуры DOM. Цель тестов - уверенность, что форма работает, а не проверка деталей реализации.

Давайте, посмотрим **как надо написать этот тест**:

```
// GOOD EXAMPLE
test('shows success message after submitting the form', async () => {
  // 1. Setup (User Event)
  const user = userEvent.setup();
  render(<LoginForm />);

  // 2. Находим элементы так, как их видит пользователь (Role, Label)
  const input = screen.getByLabelText(/username/i);
  const button = screen.getByRole('button', { name: /submit/i });

  // 3. Взаимодействие
  await user.type(input, 'myUser');
  await user.click(button);

  // 4. Assertion (Ожидание результата)
  // Используем findBy для асинхронного ожидания появления элемента
  const successMsg = await screen.findByText(/welcome back/i);
  expect(successMsg).toBeInTheDocument();
});
```

Мы ищем по **Label** («Username») и **Role** («button», «Submit»). Пока пользователь видит поле с подписью «Username» и кнопку «Submit», тест будет проходить, даже если вы полностью перепишете верстку или смените классы.

Использование **getByRole** и **getByLabelText** - это скрытая проверка доступности вашего приложения. Если тест падает на строке **screen.getByRole('button')**, это значит, что не только тест не может найти кнопку, но и **скринридер** (программа для незрячих людей) её не увидит.

Здесь используется **user-event** (user.type, user.click) вместо устаревшего **fireEvent**.

Использование await **screen.findByText(...)** убирает необходимость в ручных костылях вроде **setTimeout** или **waitFor**. Когда мы нажимаем «Submit», приложению нужно время (сходить в API, обновить стейт, перерисовать компонент). Сообщение "Welcome back" появляется не мгновенно.

Метод findBy (в отличие от getBy) автоматически «ждет» и перепроверяет наличие элемента в течение определенного тайм-аута (по дефолту 1000мс). Это делает код чище и линейнее.

Этот тест лучше, потому что он отвязан от кода и привязан к UX. Он проверяет, работает ли приложение для человека, а не просто наличие тегов в HTML.